#2

IN THE UNITED STATES PATENT OFFICE

Applicant: Komatsu et al
Serial No.: unassigned
Filed: herewith

Date: October 29, 2001
Docket No.: JP920000285
Group Art Unit: unassigned
Examiner: unassigned

For: Program Optimization

SUBMISSION OF CERTIFIED PRIORITY DOCUMENT

To the Assistant Commissioner of Patents:

Enclosed herewith is a certified copy of the above-identified Application No. 2000-331427, filed in Japan on October 30, 2000, for which applicant claims priority under 35 U.S.C. & 119.

It is requested that this be made part of record in the subject application.

Respectfully submitted,

By: Lauren Bruzzone
Lauren Bruzzone
(Reg No 35,082)
Telephone:(914)945-3255

IBM CORPORATION
Intellectual Property Law Dept.
T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, N.Y. 10598
(914) 945-3255

日　本　国　特　許　庁
JAPAN PATENT OFFICE

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed with this Office

出願年月日
Date of Application: ２０００年１０月３０日

出願番号
Application Number: 特願２０００－３３１４２７

出願人
Applicant(s): インターナショナル・ビジネス・マシーンズ・コーポレーション

２００１年　６月１９日

特許庁長官
Commissioner,
Japan Patent Office 及川耕造

出証番号　出証特２００１－３０５７７５１

【書類名】　特許願

【整理番号】　JP9000285

【提出日】　平成12年10月30日

【あて先】　特許庁長官　殿

【国際特許分類】　G06F　5/06

【発明者】

【住所又は居所】　神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　東京基礎研究所内

【氏名】　稲垣　達氏

【発明者】

【住所又は居所】　神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　東京基礎研究所内

【氏名】　小松　秀昭

【特許出願人】

【識別番号】　390009531

【氏名又は名称】　インターナショナル・ビジネス・マシーンズ・コーポレーション

【代理人】

【識別番号】　100086243

【弁理士】

【氏名又は名称】　坂口　博

【代理人】

【識別番号】　100091568

【弁理士】

【氏名又は名称】　市位　嘉宏

【代理人】

【識別番号】　100106699

【弁理士】

【氏名又は名称】　渡部　弘道

【復代理人】

【識別番号】　100104880

【弁理士】

【氏名又は名称】　古部　次郎

【選任した復代理人】

【識別番号】　100100077

【弁理士】

【氏名又は名称】　大場　充

【手数料の表示】

【予納台帳番号】　081504

【納付金額】　21,000円

【提出物件の目録】

【物件名】　明細書　1

【物件名】　図面　1

【物件名】　要約書　1

【包括委任状番号】　9706050

【包括委任状番号】　9704733

【包括委任状番号】　0004480

【プルーフの要否】　要

【書類名】　　明細書

【発明の名称】　　プログラムの最適化方法及びこれを用いたコンパイラ

【特許請求の範囲】

【請求項１】　プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行う最適化方法において、

処理対象である前記プログラムを解析し、例外が発生する可能性のある例外発生可能命令と、当該例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出するステップと、

検出された前記例外発生検出命令を、前記例外の発生条件を検出する第１の命令と、前記例外の発生条件が検出された場合に処理を前記例外処理へ条件分岐させる第２の命令とに分割するステップと、

前記第１の命令に対して、前記例外の発生条件を検出した場合に前記第２の命令に移行し、前記例外発生条件を検出しなかった場合に前記例外発生可能命令に移行するように依存関係を設定するステップと

を含むことを特徴とするプログラムの最適化方法。

【請求項２】　前記命令の依存関係を設定するステップの後に、

プログラムの所定の範囲における複数の前記例外発生検出命令に基づいて得られた複数の前記第２の命令をまとめるステップをさらに含み、

複数の前記例外発生検出命令に基づいて得られた複数の前記第１の命令が例外の発生条件を検出したことを一括して判断する、

請求項１に記載のプログラムの最適化方法。

【請求項３】　前記命令を分割するステップは、前記第１の命令として、前記例外の発生条件を検出した場合にフラグを立てる命令を生成するステップを含み、

前記第２の命令をまとめるステップは、複数の前記例外発生検出命令に基づいて得られた複数の前記第１の命令において、少なくともいずれか一つが前記フラグを立てている場合に、例外が発生したことを検出し、前記第２の命令に処理を移行させる命令を生成するステップを含む、

請求項２に記載のプログラムの最適化方法。

【請求項４】　前記例外発生可能命令を検出するステップの後に、

前記例外発生可能命令が副作用を伴う場合、前記例外発生検出命令の分割前における当該副作用に関する順序制約を保存する補償コードを生成するステップをさらに含む、

請求項１に記載のプログラムの最適化方法。

【請求項５】　前記命令の依存関係を設定するステップの後に、

前記第１の命令を条件分岐とし、当該条件分岐を反映するようにプレディケートの割付を行うステップをさらに含む、

請求項１に記載のプログラムの最適化方法。

【請求項６】　前記命令の依存関係を設定するステップの後に、

前記第１の命令を条件分岐とし、コードスケジューリングの結果に応じて、前記第１の命令の分岐先に、前記第１、第２の命令に分割される前の前記例外発生検出命令に関する順序制約を満足するように補償コードを生成するステップをさらに含む、

請求項１に記載のプログラムの最適化方法。

【請求項７】　プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行うコンパイラにおいて、

処理対象である前記プログラムを解析し、当該プログラム中の演算子の依存関係を示すグラフを生成するグラフ生成部と、

生成された前記グラフを編集し、例外による前記演算子の順序制約を緩和するグラフ編集部と、

編集された前記グラフにおける前記演算子の依存関係を反映させたプログラムコードを生成するコード再生部とを備え、

前記グラフ編集部は、

前記グラフ中から、例外が発生する可能性のある例外発生可能命令と、当該例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出し、

検出された前記例外発生検出命令を、前記例外の発生条件を検出する第１の命令と、前記例外の発生条件が検出された場合に処理を前記例外処理へ条件分岐さ

せる第２の命令とに分割し、

前記グラフ上の前記第１の命令に対して、前記例外の発生条件を検出した場合に前記第２の命令に移行し、前記例外発生条件を検出しなかった場合に前記例外発生可能命令に移行するように依存関係を設定すること

を特徴とするコンパイラ。

【請求項８】　前記グラフ編集部は、

前記グラフから、前記例外発生検出命令の分割前における前記例外発生可能命令に関する順序制約と、前記例外発生検出命令の分割前における当該例外発生検出命令に先行する順序制約とを取り除く、

請求項７に記載のコンパイラ。

【請求項９】　前記グラフ編集部は、

プログラムの所定の範囲における前記グラフ上の複数の前記例外発生検出命令に基づいて得られた複数の前記第２の命令を合成する、

請求項７に記載のコンパイラ。

【請求項１０】　前記グラフ編集部は、

前記例外発生可能命令が副作用を伴う場合に、前記例外発生検出命令の分割前における当該副作用に関する順序制約を保存するように、前記グラフ上の前記例外発生可能命令に対して順序制約を設定する、

請求項７に記載のコンパイラ。

【請求項１１】　前記グラフ編集部は、

前記第１の命令を条件分岐とし、コードスケジューリングの結果に応じて、前記グラフにおける前記第１の命令の分岐先に、前記第１、第２の命令に分割される前の前記例外発生検出命令に関する順序制約を満足するように補償コードを生成する、

請求項７に記載のコンパイラ。

【請求項１２】　プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行うコンパイラにおいて、

前記ソースコードを編集可能な中間コードに変換する中間コード生成部と、

前記中間コードに対して最適化を行う最適化部と、

最適化された前記中間コードから機械語コードを生成する機械語コード生成部とを備え、

前記最適化部は、

処理対象である前記プログラムの中間コードを解析し、当該プログラムの所定の範囲において、例外が発生する可能性のある例外発生可能命令を他の命令に先んじて実行するように、当該プログラムを変形し、

前記例外発生可能命令において例外が発生した場合に当該例外発生可能命令以降の命令が実行されないように、前記プログラムにおける命令間の依存関係を設定し、

前記プログラムの所定の範囲において、複数存在する前記例外発生可能命令のうち少なくとも一つが例外を発生する場合に、当該例外の発生を検出し、当該例外に対応する例外処理へ移行するように、前記プログラムを変形すること

を特徴とするコンパイラ。

【請求項１３】　前記最適化部は、

前記例外発生可能命令が副作用を伴う場合に、前記プログラムの変形前における当該副作用に関する順序制約を保存するように、前記グラフ上の前記例外発生可能命令に対して順序制約を設定する、

請求項１２に記載のコンパイラ。

【請求項１４】　プログラムのソースコードを入力する入力装置と、

入力された前記プログラムをコンパイルして機械語コードに変換するコンパイラと、

機械語コードに変換された前記プログラムを実行する処理装置とを備えたコンピュータ装置において、

前記コンパイラは、

処理対象である前記プログラムを解析し、例外が発生する可能性のある例外発生可能命令と、当該例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出し、

検出された前記例外発生検出命令を、前記例外の発生条件を検出する第１の命令と、前記例外の発生条件が検出された場合に処理を前記例外処理へ条件分岐さ

せる第２の命令とに分割し、

前記第１の命令に対して、前記例外の発生条件を検出した場合に前記第２の命令に移行し、前記例外発生条件を検出しなかった場合に前記例外発生可能命令に移行するように依存関係を設定すること

を特徴とするコンピュータ装置。

【請求項１５】　前記コンパイラは、

プログラムの所定の範囲における複数の前記例外発生検出命令に基づいて得られた複数の前記第１の命令が例外の発生条件を検出したことを一括して判断するように、複数の当該例外発生検出命令に基づいて得られた複数の前記第２の命令をまとめる、

請求項１４に記載のコンピュータ装置。

【請求項１６】　コンピュータに実行させるプログラムを当該コンピュータの入力手段が読取可能に記憶した記憶媒体において、

前記プログラムは、

処理対象であるプログラムを解析し、例外が発生する可能性のある例外発生可能命令と、当該例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出する処理と、

検出された前記例外発生検出命令を、前記例外の発生条件を検出する第１の命令と、前記例外の発生条件が検出された場合に処理を前記例外処理へ条件分岐させる第２の命令とに分割する処理と、

前記第１の命令に対して、前記例外の発生条件を検出した場合に前記第２の命令に移行し、前記例外発生条件を検出しなかった場合に前記例外発生可能命令に移行するように依存関係を設定する処理と

を前記コンピュータに実行させる記憶媒体。

【請求項１７】　前記記憶媒体に記憶される前記プログラムは、前記命令の依存関係を設定する処理の後に、

プログラムの所定の範囲における複数の前記例外発生検出命令に基づいて得られた複数の前記第２の命令をまとめる処理をさらに含み、

複数の前記例外発生検出命令に基づいて得られた複数の前記第１の命令が例外

の発生条件を検出したことを一括して判断する、
請求項１６に記載の記憶媒体。

【請求項１８】　コンピュータに、

処理対象であるプログラムを解析し、例外が発生する可能性のある例外発生可能命令と、当該例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出する処理と、検出された前記例外発生検出命令を、前記例外の発生条件を検出する第１の命令と、前記例外の発生条件が検出された場合に処理を前記例外処理へ条件分岐させる第２の命令とに分割する処理と、前記第１の命令に対して、前記例外の発生条件を検出した場合に前記第２の命令に移行し、前記例外発生条件を検出しなかった場合に前記例外発生可能命令に移行するように依存関係を設定する処理とを実行させるプログラムを記憶する記憶手段と、

前記記憶手段から前記プログラムを読み出して当該プログラムを送信する送信手段とを備えたことを特徴とするプログラム伝送装置。

【請求項１９】　前記記憶手段に記憶される前記プログラムは、前記命令の依存関係を設定する処理の後に、

プログラムの所定の範囲における複数の前記例外発生検出命令に基づいて得られた複数の前記第２の命令をまとめる処理をさらに含み、

複数の前記例外発生検出命令に基づいて得られた複数の前記第１の命令が例外の発生条件を検出したことを一括して判断する、
請求項１８に記載のプログラム伝送装置。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】

本発明は、コンピュータプログラムの最適化方法に関し、特に例外を起こす可能性のある命令（以下、例外発生可能命令と称す）を含むプログラムに対して命令レベルでの並列性を得るための最適化方法に関する。

【０００２】

【従来の技術】

通常、プログラミング言語で記述されたプログラムのソースコードをコンパイルする際、コンピュータにおける実行速度の向上を図るために、当該プログラムの最適化を行っている。

最適化の手法には種々の方法があるが、米国インテル社及び米国ヒューレット・パッカード社によるＩＡ－６４のアーキテクチャに対応したＣＰＵのように、命令レベルの並列実行が可能なプロセッサ上で実行されることを前提として、命令レベルでの並列性を抽出するためのコードスケジューリングにおいて、できるだけ並行して発行可能な命令が多くなるように、命令の順番の入れ替えを行うという最適化がある。

【０００３】

ところがプログラム中には、順番の入れ替えに対する様々な制約があり、コードスケジューリングの働きは抑えられている。順番の入れ替えに対する制約には、コントロールによる制約、メモリアクセスによる制約、例外による制約がある。これらの制約のうち、コントロールによる制約とメモリアクセスによる制約に対しては、ハードウェアサポートによって緩和する機構が実用化されている。しかし、例外による制約をハードウェアサポートによって緩和する機構は実現されていない。

【０００４】

また、プログラムにおける命令レベルの並列性を抽出した場合、先行的に発行された命令が起こした例外を処理することが必要である。従来のこの種の処理の代表的な方法は、そのような命令が上げた例外を即時には実行せず、最適化を行う前のプログラムにおいてその命令がもともと置かれていた場所で例外発生の有無を判定し、実際の例外処理を行う方法である。これによって、例外発生時にもオリジナルのコンテクストを保存しながら、例外発生可能命令を先行的に実行できる。

【０００５】

ところで、例外が発生する可能性のある命令を投機的に実行して並列性を抽出するため、従来の技術は、投機的に実行される例外発生可能命令で起きる例外（以下、このような例外を投機的例外と称す）を抑制するハードウェアサポートを

仮定していた。

この種の従来技術としては、例えば下記の文献１に開示された技術がある。

文献１：Scott A. Mahlke, William Y. Chen, Roger A. Bringmann, Richard E. Hank, Wen-Mei W. Hwu, B. Ramakrishna Rau, and Michael S. Schlansker, "Sentinel Scheduling: A Model for Compiler-Controlled Speculative Execution", ACM Transactions on Computer Systems, Vol. 11, No. 4, November 1993, Pages 376-408

【０００６】

文献１は、ハードウェアサポートの元に、コンパイラが投機的な命令移動と例外発生時の補償コードの生成を行う方法を提案している。ここで仮定しているハードウェアでは、投機的な命令に関しては例外を上げる代わりに特別な演算結果を生成し、その値が後続の投機的な命令によって推移的に伝搬される。そして、投機的でない命令はこの特別な値を使用したときに例外を発生する。実際に例外が起きたかどうかを確認する命令はsentinel instructionと呼ばれる。

【０００７】

この従来技術では、以下の手順で例外発生可能命令を投機的にスケジューリングする。

１．各例外発生可能命令について、生成される値を最後に使用する命令(potential sentinel instruction)を求める。

２．リストスケジューリングによって、各命令の最も早い実行開始時刻を求める。例外発生可能命令が分岐、副作用（メモリへの書き込み）、他の例外発生可能命令を越えてスケジューリングされた場合、その命令は投機的である。したがって、手順１で求めた情報を使って、必要な場合は明示的なsentinel instructionを生成する。

３．例外状態からの回復をソフトウェアで行う場合は、各sentinel instructionに対応した例がハンドラに対して例外発生時の補償コードを生成する。

【０００８】

また、この種の他の従来技術として、特開平１２－０２０３２０号公報に開示

された技術がある。同公報は、コンパイラおよびランタイムライブラリのサポートによって例外発生可能命令の投機的な実行を実現する手法を開示している。

【０００９】

この従来技術では、以下の手順で例外発生可能命令の投機的実行を行う。

１．コンパイラが、最終的に生成されるオブジェクトコード中で、例外発生可能命令の投機的実行によって実行順序が元のプログラムと変わった部分を割り込み禁止区間として登録する。

２．例外発生時に、例外ハンドラが割り込み禁止区間情報を元にして、補償コードを動的に生成し実行する。

【００１０】

【発明が解決しようとする課題】

上述したように、プログラムを命令レベルで並列実行させるためのコードスケジューリングには種々の制約があり、特に例外による制約ハードウェアサポートによって緩和する機構は実現されていない。

これに対し、Ｊａｖａのような、例外の取り扱いが厳密でなければならない言語においては、例外発生可能命令間の制約が、プログラムにおける命令レベルの並列性の抽出に及ぼす影響が大きい。このため、Ｊａｖａなどの例外の取り扱いが厳密でなければならない言語で記述されたプログラムに対するコンパイラにおいて、命令レベルの並列性を抽出することは困難であった。

【００１１】

Ｊａｖａ言語を例としてさらに詳細に説明する。

Ｊａｖａ言語の仕様は、プログラムの実行中に発生する例外条件の厳密な取り扱いを定めている。システムで定められている例外状態（オブジェクトのポインタがｎｕｌｌである場合、配列の長さを超えた添字でアクセスした場合、整数の０除算を行った場合など）及びユーザーが定義した例外状態は、全て例外の発生を実行時に検出して、ユーザーにより定義された例外ハンドラで処理することができる。そして、例外ハンドラの実行が終了すると制御は例外を検出した領域から脱出する。

【００１２】

ここで、例外の発生は制御フローを変更するので、プログラム中の副作用（配列の要素やクラス、オブジェクトのフィールドへの書き込み）の発生と例外の発生の間には全順序がある。すなわち、ユーザー定義の例外ハンドラが実行される時点では、プログラム中で次の二つの条件を満足しなければならない。

１．例外の発生より先に行われた副作用は全て観測されなければならない。

２．例外の発生より後に行われる副作用は観測されてはならない。

【００１３】

Ｊａｖａでは、配列の要素やオブジェクトのフィールドへの読み出し及び書き込みなどの基本的な操作が実行時の例外を伴う。すなわち、Ｊａｖａプログラムが機械語命令にコンパイルされると、これらの操作は実行時の例外を検出する命令と、実際の操作を行うメモリアクセスや演算に変換される。そして、メモリアクセスや演算は例外を発生する可能性がある。

ここで、発生する例外は、アクセス違反やゼロで割る除算などを実行することによってプロセッサ（ハードウェア）が取る状態としての例外である。プログラムの実行に伴う例外としては、この他、プログラム言語の仕様外の入力に対するプログラムの実行状態としての例外がある。以下の説明において、特に前者の例外を他の例外と区別する必要がある場合は、ハードウェア例外と称する。

ハードウェア例外は、機械語プログラムの間違いによって、プロセッサが不正なメモリ領域に読み書きを行ったり、計算できない演算を行ったりしたために、正しく実行を継続できない場合に発生する。一方、その他の例外は、ユーザーの書いたプログラムの間違いにより、配列のサイズを超えた要素を読み書きしたり関数に正しくない入力値を与えたりした場合のエラー処理として、言語及びプログラムが定める例外である。

【００１４】

このハードウェア例外の発生を回避するため、コードスケジューリングにおいて、副作用を行う命令、例外発生を検出する命令、例外発生可能命令の間に実行順序の制約が生じる。

Ｊａｖａプログラムにおいては、この実行順序の制約を伴う配列要素のアクセスやフィールド変数のアクセスが頻繁に現れるため、これらの順序制約が、コー

ドスケジューリングにおいて命令レベルの並列性を抽出する処理の大きな妨げになっていた。

【００１５】

また、上述したように、プログラムにおける命令レベルの並列性を抽出した場合、先行的に発行された命令が起こした例外を処理することが必要であるが、従来は、そのような命令が上げた例外を即時には実行せず、最適化を行う前のプログラムにおいてその命令がもともと置かれていた場所で例外発生の有無を判定し、実際の例外処理を行うことにより対応していた。

【００１６】

しかしこの場合、先行的に上げられた例外の発生の有無を判定する命令のコストを無視することはできない。例えば上述したＩＡ－６４では、投機的例外を検出するための特別な機械語が用意されているが、大規模に命令の移動を行った場合、そのような例外の発生を判定するための命令が実行速度に与える影響は大きくなるものと考えられる。

【００１７】

文献１に記載された従来技術では、ハードウェア例外を発生する可能性のある命令を投機的に実行することはできるが、副作用や例外発生を検出する命令の間には順序制約が残る。副作用のある命令を投機的に実行するためには、さらにメモリへの書き込みをキャンセルするためのハードウェアコストが必要である。

また、この従来技術は、ハードウェアサポートを必要とするため、投機的実行を行うハードウェアを持たない一般のプロセッサには直接適用できない。

さらに、ソフトウェアによりこの従来技術による方法をエミュレーションする手法は付加的な命令によるコストが高く現実的ではない。

したがって、この従来技術は、単純なハードウェア例外発生可能命令の投機実行と、例外状態の回復によるモデルではオーバーヘッドが高く、Ｊａｖａ言語のコンパイルにおける例外発生可能命令の順序制約を緩和するには十分ではないと言える。

【００１８】

また、特開平１２－０２０３２０号公報に記載された従来技術では、例外が発

生しないときにはソフトウェアオーバーヘッドなしで実現できる。しかし、補償コードを例外ハンドラで実行時に生成するため、機械語命令列が全て決定された後の最適化、すなわちレジスタ割付後のコードスケジューリングにおける最適化にしか適用できない。このため、コード変形を伴う最適化で、例外発生可能命令の投機的な実行による並列性を利用することができない。

【００１９】

そこで、本発明は、例外発生可能命令の他の命令への先行制約をソフトウェア的に緩和することによって、例外発生可能命令を含むプログラムの命令レベル並列性を効果的に得ることを目的とする。

【００２０】

【課題を解決するための手段】

かかる目的のもと、本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行う最適化方法において、処理対象のプログラムを解析し、例外発生可能命令と、例外発生検出命令とを検出するステップと、検出された例外発生検出命令を、例外の発生条件を検出する第１の命令と、例外の発生条件が検出された場合に処理を例外処理へ条件分岐させる第２の命令とに分割するステップと、第１の命令に対して、例外の発生条件を検出した場合に第２の命令に移行し、例外発生条件を検出しなかった場合に例外発生可能命令に移行するように依存関係を設定するステップとを含むことを特徴とする。

このようにプログラムを変形し、依存関係を設定することによって、例外発生可能命令における順序制約を緩和することができる。

【００２１】

ここで、この最適化方法は、命令の依存関係を設定するステップの後に、プログラムの所定の範囲における複数の例外発生検出命令に基づいて得られた複数の第２の命令をまとめるステップをさらに含む。これにより、複数の例外発生検出命令に基づいて得られた複数の第１の命令が例外の発生条件を検出したことを一括して代表的な判断を行う。

【００２２】

さらに詳しくは、この最適化方法において、命令を分割するステップは、第１の命令として、例外の発生条件を検出した場合にフラグを立てる命令を生成するステップを含み、第２の命令をまとめるステップは、複数の例外発生検出命令に基づいて得られた複数の第１の命令において、少なくともいずれか一つがフラグを立てている場合に、例外が発生したことを検出し、第２の命令に処理を移行させる命令を生成するステップを含む。

上述したプログラムの所定の範囲として基本ブロックを用いることができる。そして、この基本ブロック内における例外の発生の判定を一つのフラグを用いてまとめて行うことにより、実際の例外発生を検出するソフトウェアオーバーヘッドを一つの分岐命令で済ませることができる。

【００２３】

また、この最適化方法は、例外発生可能命令を検出するステップの後に、例外発生可能命令が副作用を伴う場合、例外発生検出命令の分割前におけるこの副作用に関する順序制約を保存する補償コードを生成するステップをさらに含む。

ここで、副作用とは、命令の実行に伴って行われる所定のデータのメモリへの書き込みである。

【００２４】

さらに、この最適化方法は、コンパイル後のプログラムを実行するコンピュータ装置のＣＰＵがプレディケート付き命令を実行する機能を持つ場合、例外発生可能命令をガードするため、命令の依存関係を設定するステップの後に、第１の命令を条件分岐とし、この条件分岐を反映するようにプレディケートの割付を行うステップをさらに含む。

【００２５】

これに対し、この最適化方法は、コンパイル後のプログラムを実行するコンピュータ装置のＣＰＵがプレディケート付き命令を実行する機能を持たない場合、例外発生可能命令をガードするため、命令の依存関係を設定するステップの後に、第１の命令を条件分岐とし、コードスケジューリングの結果に応じて、第１の命令の分岐先に、第１、第２の命令に分割される前の例外発生検出命令に関する順序制約を満足するように補償コードを生成するステップをさらに含む。

【００２６】

また本発明は、上述した最適化方法を含むコンパイルを、コンピュータ装置に実行させるコンピュータプログラムとして作成し、このコンピュータプログラムを格納した記憶媒体や、このコンピュータプログラムを伝送する伝送装置として提供することができる。

さらに本発明は、このコンピュータプログラムを実行するコンパイラを備えたコンピュータ装置として提供することができる。

【００２７】

また、本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行うコンパイラにおいて、処理対象のプログラムを解析し、このプログラム中の演算子の依存関係を示すグラフを生成するグラフ生成部と、生成されたこのグラフを編集し、例外による演算子の順序制約を緩和するグラフ編集部と、編集されたこのグラフにおける演算子の依存関係を反映させたプログラムコードを生成するコード再生部とを備え、グラフ編集部は、このグラフ中から、例外が発生する可能性のある例外発生可能命令と、この例外の発生条件を検出し例外が発生した場合に例外処理への分岐を行う例外発生検出命令とを検出し、検出された例外発生検出命令を、この例外の発生条件を検出する第１の命令と、この例外の発生条件が検出された場合に処理を例外処理へ条件分岐させる第２の命令とに分割し、このグラフ上の第１の命令に対して、例外の発生条件を検出した場合に第２の命令に移行し、例外発生条件を検出しなかった場合に例外発生可能命令に移行するように依存関係を設定することを特徴とする。

【００２８】

ここで、このグラフ編集部は、このグラフから、例外発生検出命令の分割前における例外発生可能命令に関する順序制約と、例外発生検出命令の分割前におけるこの例外発生検出命令に先行する順序制約とを取り除く。

これにより、例外発生可能命令に対する順序制約をプログラムの所定の領域（基本ブロックなど）におけるクリティカルパス上から除去することができる。

【００２９】

また、本発明は、プログラミング言語で記述されたプログラムのソースコードを機械語に変換し、プログラムの最適化を行うコンパイラにおいて、このソースコードを編集可能な中間コードに変換する中間コード生成部と、この中間コードに対して最適化を行う最適化部と、最適化された中間コードから機械語コードを生成する機械語コード生成部とを備え、この最適化部は、処理対象であるプログラムの中間コードを解析し、プログラムの所定の範囲において、例外が発生する可能性のある例外発生可能命令を他の命令に先んじて実行するように、このプログラムを変形し、この例外発生可能命令において例外が発生した場合にこの例外発生可能命令以降の命令が実行されないように、このプログラムにおける命令間の依存関係を設定し、このプログラムの所定の範囲において、複数存在する例外発生可能命令のうち少なくとも一つが例外を発生する場合に、この例外の発生を検出し、この例外に対応する例外処理へ移行するように、このプログラムを変形することを特徴とする。

【００３０】

さらにここで、この最適化部は、この例外発生可能命令が副作用を伴う場合に、このプログラムの変形前におけるこの副作用に関する順序制約を保存するように、このグラフ上の例外発生可能命令に対して順序制約を設定する。

【００３１】

【発明の実施の形態】

以下、添付図面に示す実施の形態に基づいて、この発明を詳細に説明する。

まず、本発明の概要について説明する。本発明は、例外を起こす可能性のある命令（例外発生可能命令）における他の命令に対する先行制約をソフトウェア的に緩和することによって、例外発生可能命令を含むプログラムの命令レベル並列性を得る。

【００３２】

本発明において、コンパイラは、他の命令に先んじて発行された例外発生可能命令がソフトウェアによってガードされるようにコードの変形を行う。そして、投機的に実行される例外発生可能命令により例外（投機的例外）が起きる場合、コンパイラが用意した例外ハンドラに分岐し、必要な副作用（メモリへの書き込

み)及び例外の検出を行ってから例外の発生を表すフラグを立て、ガードされていない命令のコンテクストに復帰して実行を再開する。このように、コンパイラにおいて例外の発生をコントロールすることにより、より多くの種類の命令を先行的に発行できる。

その後、例外発生可能命令が元々配置されていたコンテクストにおいて、上記の例外の発生を表すフラグを判定し、フラグが立っていれば、当該例外に対する固有の例外処理を行う。

このように、投機的例外の判定を条件分岐やプレディケートなどのソフトウェアで行うことにより、例外発生可能命令は、理論的に、コントロールによる制約、メモリアクセスによる制約、例外による制約の全てに制限されずに先行的に発行できるようになる。

【００３３】

しかしながら、このような投機的例外を判定するための命令を実行するコストは無視できるものではない。さらに、ハードウェアでサポートされていない投機処理を行う場合は、この例外判定をソフトウェアによって行うこととなるため、そのコストは大きくなる。

【００３４】

そこで、本発明は、プログラムにおける基本ブロックの先頭において、その基本ブロック内におけるフラグのいずれか一つでも立っていれば、これを一括して検出する代表的な判定を用意する。この判定は、フラグをビットワイズに表現することによって、機械語１命令で高速に処理できる。

この代表的な判定において、基本ブロック内のいずれかのフラグが立っていると判明した場合、まず、オリジナルのプログラムで記述されているコンテクストを回復させるのに必要十分な命令を実行する。そして、立っているフラグを検出し、当該フラグに対応する例外に対する処理を行う。

なお、基本ブロックとは、ストレートコード、すなわちコントロールフローが途中に入ることもなく、途中から出ることもないようなコード列の範囲をブロックで示したものである。

【００３５】

図１は、本発明の実施の形態におけるコンパイラの構成を説明する図である。

本実施の形態では、処理対象である、例外の取り扱いが厳密でなければならない言語としてＪａｖａを用いる例について説明する。すなわち、最適化の対象であるプログラムをＪａｖａプログラムとし、コンパイラをＪａｖａのJust In Time Compilerとする。したがって、図１に示す本実施の形態のコンパイラ１００は、Ｊａｖａバイトコードの形式で記述されたプログラムを入力してコンパイルし、当該プログラムを実行する計算機に対応した機械語コード形式に変換して出力する。ただし、本発明を、他の種々のプログラム言語で記述されたプログラムに対するコンパイラに適用できることは言うまでもない。

また、コンパイラ１００として、ＪａｖａのJust In Time Compilerを想定する場合、コンパイラ１００は、当該プログラムを実行するコンピュータ装置に搭載されることとなる。すなわち、図示しないがこのコンピュータ装置は、プログラムのソースコードを入力する入力装置と、コンパイラ１００を実現するコンピュータプログラムを格納したメモリと、メモリに格納されたコンピュータプログラムの制御によりコンパイラ１００としての処理を実行すると共に、コンパイラ１００により機械語コードに変換されたプログラムを実行する計算機（ＣＰＵ）とを備える。また、このコンピュータ装置は、コンパイラ１００を実現するコンピュータプログラムを、上述したＣＤ－ＲＯＭやフロッピーディスクから読み出すディスクドライブや、ネットワークを介して受信する受信部を備える。

【００３６】

図１を参照すると、本実施の形態のコンパイラ１００は、四つ組中間コード生成部１０と、四つ組中間コード最適化部２０と、機械語コード生成部３０とを備える。

上記構成において、四つ組中間コード生成部１０は、Ｊａｖａバイトコード形式で与えられたプログラムを解析し、四つ組形式で表現された中間コード（以下、四つ組中間コードと称す）に変換する。

四つ組中間コード最適化部２０は、四つ組中間コード生成部１０により生成された四つ組中間コードに対して、すなわち、最終的に生成される機械語コードの実行時間が短くなるように、計算の冗長性を取り除いたり、中間コードを移動し

たりする。この最適化の過程で、プログラム中の例外発生可能命令を投機的に実行するように命令の順序の入れ替えを行う。

機械語コード生成部３０は、最適化された四つ組中間コードを、当該プログラムを実行する計算機に対応した機械語コード形式に変換し出力する。

【００３７】

本実施の形態は、Ｊａｖａ言語の例外処理をコンパイルする際の以下のような特徴に注目して、ソフトウェアによる例外発生可能命令の投機実行を効率的に行う。

１．例外処理は必ず制御ブロックからの脱出である。また、ユーザー定義の例外ハンドラで検出できるのは副作用の結果だけであり、それ以外の中間結果は例外ハンドラ以降の実行では参照されない。したがって、投機的に実行した例外発生可能命令が例外を起す場合は、正確に全ての状態を回復する必要はなく、副作用の順序を保証し、例外の種類と引数を特定できれば良い。

２．例外はプログラムにおける「例外的条件」を扱うので、プログラムの実行全体から見ると例外が発生する頻度は比較的低いと仮定できる。したがって、例外が発生した場合の処理のソフトウェアオーバーヘッドはある程度許容できる。

３．本発明は、Ｊａｖａチップなどではない通常のプロセッサを対象としてコンパイルを行う場合、例外の検出のためにソフトウェアによる明示的な命令列によるオーバーヘッドを伴っている。代表的かつ最も頻繁に現れる例外は、配列の長さを超えたインデクスによるアクセスを検出する例外（ArrayIndexOutOfBoundsException）である。そこで、配列サイズとインデクス変数をチェックする既存の命令を使って、投機的例外の発生を検出することができる。

【００３８】

Ｊａｖａ言語における以上の特徴を利用して、Ｊａｖａ言語の例外発生可能命令をソフトウェアで投機的に実行するために以下の基本方針を採用する。

１．Ｊａｖａ言語における例外を検出する既存の命令列を利用して、ハードウェア例外発生可能命令の実行を、条件分岐やプレディケート付き命令によってソフトウェア的にガードする。すなわち、ハードウェア例外が発生する場合には当該例外発生可能命令は実行されない。これによって、例外が発生しない場合の実行

において付加的な命令オーバーヘッドを導入しなくて済むこととなる。ガードされた例外発生可能命令は、ガードに対して制御依存を持つ代わりに副作用や他の例外発生可能命令に対して順序制約を持たなくなるので、例外発生可能命令による順序制約が緩和される。

２．Ｊａｖａ言語の定める例外が発生したかどうかを検出し、ユーザー定義の例外ハンドラに分岐する命令と例外発生条件を検出する命令とを分離し、基本ブロック内で例外が起ったことを表す代表のフラグを用いる。これによって、実際に例外ハンドラに条件分岐する処理を基本ブロック内で一つにまとめることができるので、実際の例外発生を検出するソフトウェアオーバーヘッドを一つの分岐命令で済ませることができる。

３．例外発生検出時の補償コードを、例外発生検出命令の分岐先（または、プレディケート付き命令における負の条件のプレディケートによる分岐先）に生成する。これによって補償コードのための余分な分岐命令を通常実行されるパスに挿入することを回避することができる。例外処理は必ず脱出であるため、補償コードは副作用及び例外の検出によって構成される。補償コードを生成するために、副作用や例外発生可能命令の間にある順序制約を利用する。

【００３９】

次に、本実施の形態における例外発生可能命令の投機的実行、副作用の投機的実行、補償コードの生成の手順を説明する。

プログラムは、プログラム中の演算子の依存関係を示す無閉路有向グラフ（Directed Acyclic Graph；以下、ＤＡＧと略す）として表現される。ＤＡＧの頂点は演算子を表し、ＤＡＧの辺（有向辺）は演算子間の依存関係を表す。依存関係は以下の３種類である。

データ依存：先行する演算子が書き込んだ値を後継する演算子が読み出すこと（フロー依存）、先行する演算子が読み出した場所に後継する演算子が書き込みを行うこと（逆依存）、及び先行する演算子が書き込んだ場所に後継する演算子が書き込みを行うこと（出力依存）を表す。

順序制約：先行する演算子の副作用が終わった後に後継する演算子の副作用が起きることを表す。

制御依存：先行する演算子による条件判定が真である場合にのみ後継する演算子が実行されることを表す。

【００４０】

例外発生可能命令の投機的実行は、プログラム中に存在する不要な順序制約をできるだけ緩和し、ＤＡＧの演算子間の並列性を抽出することを目的としている。ＤＡＧ上で、経路上の演算を全て実行し終わるまでの時間が最も長い経路のことをグラフのクリティカルパス（critical path）と呼ぶ。したがって、ＤＡＧで表された演算を全て実行し終わるまでの時間の下限は、クリティカルパスの長さに対応することとなる。

本実施の形態では、ＤＡＧのクリティカルパス上にある頂点で、例外発生検出命令への順序制約、例外発生可能命令からの順序制約による辺を選び、例外発生可能命令の投機的実行を行うようにグラフを変形する。各変換は順序制約を除去するので、クリティカルパス長を短くすることができる。この操作を、クリティカルパス上の例外発生可能命令及び例外発生検出命令の順序制約がなくなるまで繰り返すことにより、投機的実行のオーバーヘッドを極小化し、効果を極大化することができる。

【００４１】

本実施の形態において、以上の処理を実現する四つ組中間コード最適化部２０について説明する。

図２は、四つ組中間コード最適化部２０の構成を説明するブロック図である。

図２を参照すると、四つ組中間コード最適化部２０は、四つ組中間コードをＤＡＧ（Directed Acyclic Graph）に変換するＤＡＧ生成部２１と、ＤＡＧ編集部２２と、ＤＡＧを四つ組中間コードに戻す四つ組中間コード再生部２３とを備える。

ＤＡＧ生成部２１は、四つ組中間コード生成部１０により生成された四つ組中間コードを入力として受け取り、四つ組演算子間のデータ依存と順序制約を解析して、この解析結果を反映させたＤＡＧを生成する。すなわち、ＤＡＧは、四つ組中間コードで表現されたプログラム中の各命令を頂点（ノード）とし、命令間の順序制約を有向辺（パス）で表している。

ＤＡＧ編集部２２は、ＤＡＧ生成部２１により生成されたＤＡＧを入力として受け取り、例外発生可能命令を投機的に実行するようにＤＡＧの編集を行う。

四つ組中間コード再生部２３は、ＤＡＧ編集部２２により編集されたＤＡＧを入力として受け取り、四つ組中間コード形式のプログラムに再度変換して出力する。

【００４２】

図３は、ＤＡＧ編集部２２による編集処理の全体的な流れを説明するフローチャートである。

図３を参照すると、ＤＡＧ編集部２２は、まず、ＤＡＧで表現されたプログラム中のクリティカルパス上に存在する、例外による順序制約の緩和処理を行う（ステップ３０１）。この例外による順序制約の緩和によって、元のＤＡＧに存在した例外発生可能命令は、例外の検出と例外ハンドラへの条件分岐とに分割される。そして、例外の検出から例外発生可能命令に対して新たに制御依存が追加される。

【００４３】

次に、ＤＡＧ編集部２２は、例外による順序制約の緩和処理の施されたＤＡＧに対して、例外ハンドラへの条件分岐のマージ処理を行う（ステップ３０２）。この処理により、連続した例外ハンドラへの条件分岐が、代表フラグを使用した一つの条件分岐にマージされる。

【００４４】

次に、ＤＡＧ編集部２２は、プログラムを実行する計算機（以下、対象アーキテクチャと称す）が、プレディケート（predicate）付き命令が実行可能でかつ命令レベルの並列実行が可能かどうかを判断する（ステップ３０３）。

そして、対象アーキテクチャがプレディケート付き命令を並列実行する機能を持たない場合は、投機的に実行される例外に対する補償コードの生成処理を行う（ステップ３０４）。

一方、対象アーキテクチャがプレディケート付き命令を並列実行する機能を持つ場合は、ステップ３０１の例外による順序制約の緩和処理で生成された制御依存に対応してプレディケートの割付を行う（ステップ３０５）。

以上のようにして、補償コードの生成（ステップ３０４）またはプレディケートの割付（ステップ３０５）が終了した後、ＤＡＧ編集部２２による処理を終了する。

【００４５】

図４は、図３のステップ３０１における例外による順序制約の緩和処理の詳細を説明するフローチャートである。

図４を参照すると、例外による順序制約の緩和処理において、ＤＡＧ編集部２２は、まず、処理対象であるＤＡＧの頂点の中から、同じ例外発生条件を持つ例外発生可能命令ｖと例外発生検出命令ｔとの対を探す（ステップ４０１）。そして、検出された例外発生可能命令ｖと当該例外発生検出命令ｔとに対する順序制約がクリティカルパスを形成しているかどうか判断する（ステップ４０２）。

【００４６】

この順序制約がクリティカルパスを形成している場合、すなわち、この順序制約がクリティカルパス上にある場合、ＤＡＧ編集部２２は、元の、例外発生条件を検出して例外ハンドラへの分岐を行う命令ｔを、例外条件の検出だけを行う命令ｔ’と、実際に例外ハンドラに条件分岐する命令ｃとに分割する（ステップ４０３）。

【００４７】

そして次に、ＤＡＧの頂点ｔ’から頂点ｃへデータ依存の有向辺を張る（ステップ４０４）。この辺は、命令ｔ’で検出された例外発生条件を利用して命令ｃが分岐を行うことを示す。

また、頂点ｔ’から頂点ｖへ制御依存の有向辺を張る（ステップ４０５）。この辺は、命令ｔ’において例外が発生しないことが確認された場合にのみ命令ｖが実行されることを示す。

また、順序制約で頂点ｔに先行している頂点ｐから頂点ｃへ順序制約の有向辺を張る（ステップ４０６）。この辺は、元のプログラムにおいて命令ｔに先行していた全ての副作用が終了してから命令ｃの条件分岐が実行されることを示す。

【００４８】

次に、ＤＡＧ編集部２２は、命令ｖが副作用を持っているか、すなわちメモリ

に対して書き込みを行うかどうかを判断する（ステップ４０７）。そして、命令ｖが副作用を持っている場合、ＤＡＧ編集部２２は、ＤＡＧの順序制約で頂点ｔに先行している頂点ｐから頂点ｖへ順序制約の有向辺を張る（ステップ４０８）。この辺は、変換が元のプログラムにおける副作用間の順序制約を保存することを示す。

【００４９】

命令ｖが副作用を持たない場合、及びステップ４０８を終了した後、ＤＡＧ編集部２２は、ＤＡＧの順序制約で頂点ｖに後継している頂点ｓへ、頂点ｃから順序制約の有向辺を張る（ステップ４０９）。この辺は、元の命令ｖに後継する順序制約は命令ｃが実行された場合に保証されることを示す。

【００５０】

次に、ＤＡＧ編集部２２は、ＤＡＧの順序制約で頂点ｖに先行または後継している元々の辺（ステップ４０８で導入された辺は除く）を取り除く（ステップ４１０）。同様にして、順序制約で頂点ｔに先行している辺を取り除く（ステップ４１１）。これらの処理により、例外発生可能命令に対する順序制約をクリティカルパス上から除去することが可能になる。

ステップ４１１の処理が終了すると、ＤＡＧ編集部２２は、ステップ４０１に戻り、クリティカルパス上の次の例外発生可能命令による順序制約を探す。そして、クリティカルパス上に順序制約が存在しないならば、例外による順序制約の緩和処理を終了する（ステップ４０２）。

【００５１】

図５は、図３のステップ３０２における例外ハンドラへの条件分岐のマージ処理の詳細を説明するフローチャートである。

図５を参照すると、例外ハンドラへの条件分岐のマージ処理において、ＤＡＧ編集部２２は、まず、例外による順序制約の緩和処理の済んだＤＡＧのうち、順序制約の辺によって結ばれた例外ハンドラへの条件分岐の組c_1、c_2を探す（ステップ５０１）。そして、そのような条件分岐の組c_1、c_2が存在するかどうかを判断する（ステップ５０２）。

【００５２】

条件分岐の組c_1、c_2が存在する場合、ＤＡＧ編集部２２は、ＤＡＧの順序制約で条件分岐c_1に後継している頂点ｓの制御依存に対して、条件分岐c_1のガード条件を論理積によって追加する（ステップ５０３）。追加された制御依存は、条件分岐c_1がチェックしていた条件と元々命令ｓが実行される条件の両方が成立した場合にのみ命令ｓが実行されることを示す。

【００５３】

次に、ＤＡＧ編集部２２は、ＤＡＧの条件分岐c_1から頂点ｓに対する順序制約を取り除き（ステップ５０４）、条件分岐c_1と条件分岐c_2とをマージする（ステップ５０５）。マージされた条件分岐c_2では、元の条件分岐c_1と条件分岐c_2の条件の両方がテストされる。

ステップ５０５の処理が終了すると、ＤＡＧ編集部２２は、ステップ５０１に戻り、順序制約の辺によって結ばれた例外ハンドラへの条件分岐の組c_1、c_2を探す。そして、そのような条件分岐の組c_1、c_2が存在しないならば、例外ハンドラへの条件分岐のマージ処理を終了する（ステップ５０２）。

【００５４】

図６は、図３のステップ３０４における投機的に実行される例外に対する補償コードの生成処理の詳細を説明するフローチャートである。

図６を参照すると、投機的に実行される例外に対する補償コードの生成処理において、ＤＡＧ編集部２２は、まず、コードスケジューリング最適化を行い、ＤＡＧ上において例外発生可能命令と副作用の間の順序を決定する（ステップ６０１）。このコードスケジューリング最適化では、演算子の遅延時間と対象計算機の資源の使用状況を考慮して、ＤＡＧの頂点に対して最適な順序付けを行う。

【００５５】

次に、ＤＡＧ編集部２２は、順序が決定したＤＡＧの頂点のうち、例外による順序制約の緩和処理（図３のステップ３０１及び図４参照）において投機的実行の対象となった例外発生検出命令ｔに対して、元のプログラム中で推移的に先行していた頂点ｐを探す（ステップ６０２）。

【００５６】

次に、ＤＡＧ編集部２２は、ステップ６０２により得られたＤＡＧの頂点の組

ｔ、ｐのうち、ステップ６０１のコードスケジューリング最適化によって与えられた順序によって元々のプログラムの順序制約が守られない組が存在するかどうかを調べる（ステップ６０３）。

【００５７】

元々のプログラムの順序制約が守られない頂点ｔと頂点ｐの組が存在する場合、次にＤＡＧ編集部２２は、条件分岐として生成される命令ｔの分岐先に、命令ｐを補償コードとして生成する（ステップ６０４）。ＤＡＧの頂点ｔに対して先行する頂点ｐが複数存在する場合は、各頂点ｐについて、元のプログラムの順序制約を満たす順に、命令ｔの分岐先に補償コードを生成する。

ステップ６０４の処理が終了すると、ＤＡＧ編集部２２は、ステップ６０２に戻り、ＤＡＧにおいて次の順序が満たされない頂点ｔと頂点ｐの組を探す。そして、順序が満たされない頂点ｔと頂点ｐの組が存在しないならば、投機的に実行される例外に対する補償コードの生成を終了する（ステップ６０３）。

【００５８】

以上のようにしてＤＡＧ編集部２２によるＤＡＧの編集が終了した後、四つ組中間コード再生部２３が編集の済んだＤＡＧを四つ組中間コードに戻す。そして、機械語コード生成部３０が、最適化の済んだ当該四つ組中間コードを、対象アーキテクチャに対応した機械語コードに変換してコンパイルを終了する。

【００５９】

次に、Ｊａｖａプログラムの中間コードを最適化する具体的な動作例について説明する。

例として、図７（Ａ）に示すＪａｖａプログラムの中間コードを最適化する場合を考える。NullPointerExceptionのチェックがソフトウェアによって行われるプラットフォームを仮定すると、このプログラムに対応する四つ組中間コードは図７（Ｂ）に示すように表される。

図７において、NULLは配列のNullPointerExceptionを検出する操作、LENGTHは配列の長さを得る操作、SIZEはArrayIndexOutOfBoundsExceptionを検出する操作を表す。四つ組中間コード最適化部２０のＤＡＧ生成部２１により、このプログラムの持つ制約をＤＡＧで表現した様子を図８に示す。

【００６０】

次に、四つ組中間コード最適化部２０のＤＡＧ編集部２２が、図８に示すＤＡＧを編集する。

図８に示すＤＡＧにおいて、簡単のために全ての演算のコストを１とすると、このグラフのクリティカルパス長は８サイクルである。SIZE命令とNULL命令は制御を変更するので、SIZE命令からNULL命令の間に順序制約がある。このためNULL命令に依存するLENGTH、SIZE、LOADがSIZE命令に依存を持ち、データ依存による本来のクリティカルパス長を伸ばしている。

【００６１】

まず、例外による順序制約の緩和処理（図３のステップ３０１及び図４参照）において、NULL命令を投機的に実行することにより、制約を緩和する。元々のNULL命令の動作は、例外発生の検出と例外ハンドラへの分岐からなる。これを、例外発生条件の検出（同じNULL命令で表す）と、例外処理ルーチンへの分岐（CHECK命令）とに分割する。

例外発生検出命令（NULL）は、例外発生条件を検出すると、基本ブロック内で例外が発生したことを示す代表フラグを設定する。例外処理ルーチンへの分岐命令（CHECK）は、代表フラグを調べ、もし例外が発生していれば、元のプログラムで最初に発生するはずの例外ハンドラへ分岐する。

以上の処理で、代表フラグを用いることにより、複数の例外の発生を１命令で検出することができる。例外発生可能命令であるLENGTH命令は、例外発生検出命令（NULL）によってガードされる。すなわち、NULL命令が例外発生条件を検出すると、LENGTH命令は実行されない。元のNULL命令から例外ハンドラへの分岐を分離することにより、NULL命令に先行するSIZE命令による順序制約を外すことができる。

例外発生可能命令の値を使用している他の演算については、例外発生可能命令や副作用でない限りガードする必要はない。なぜなら、例外ハンドラで観測できるのは副作用と例外の発生だけだからである。

【００６２】

以上の変形を、下のSIZE命令にも同様に適用し、例外ハンドラへの条件分岐の

マージ処理（図３のステップ３０２及び図５参照）を行うことによって得られるＤＡＧを図９に示す。

図８と図９のＤＡＧを比較すると、図８のＤＡＧではクリティカルパス長が８サイクルであったが、上記の処理でCHECK命令を一つ追加することにより、図９に示すように、クリティカルパス長が５サイクルに短縮された。

図８、図９において、データ依存は実線矢印、順序制約は一点鎖線矢印、制御依存は破線矢印で表す。後述の図１１、図１２においても同様である。

【００６３】

次に、副作用を伴う例外発生可能命令の投機的実行について具体例を挙げて説明する。副作用を伴う例外発生可能命令の投機的実行を行う場合は、直前の例外発生検出命令だけでなく、先行する全ての例外発生検出命令の条件でガードする必要がある。

例として、図１０に示すＪａｖａプログラムの中間コードを最適化する場合について考える。NullPointerExceptionのチェックがソフトウェアによって行われるプラットフォームを仮定すると、このプログラムに対応する四つ組中間コードは図１０（Ｂ）に示すように表される。四つ組中間コード最適化部２０のＤＡＧ生成部２１により、このプログラムの持つ制約をＤＡＧで表現した様子を図１１に示す。

【００６４】

次に、四つ組中間コード最適化部２０のＤＡＧ編集部２２が、図１１に示すＤＡＧを編集する。

図１１に示すＤＡＧにおいて、全ての演算のコストを１とすると、このグラフのクリティカルパス長は１０サイクルである。３番目のSIZE命令を投機的に実行する場合、STORE命令による副作用は、当該STORE命令に先行する全ての例外発生検出命令における条件の積によってガードされなければならない。

【００６５】

図１１に示すＤＡＧに対し、例外による順序制約の緩和処理（図３のステップ３０１及び図４参照）及び例外ハンドラへの条件分岐のマージ処理（図３のステップ３０２及び図５参照）を行うことによって得られるＤＡＧを図１２に示す。

図１１と図１２のＤＡＧを比較すると、図１１のＤＡＧではクリティカルパス長が１０サイクルであったが、上記の処理でCHECK命令を追加することにより、図１２に示すように、クリティカルパス長が６サイクルに短縮された。

【００６６】

次に、例外発生が検出された場合の補償コードの生成について、具体的な動作例を挙げて説明する。

上述したように、例外が発生した場合には、元のプログラムで例外発生可能命令より前に実行されている副作用と例外の検出を実行しなければならない。ＩＡ－６４のようなプレディケート付き命令を実行可能なアーキテクチャでは、例外発生検出命令から例外発生可能命令及び副作用への制御依存をプレディケートで実装すれば、補償コードは不要である。この場合、代表フラグのテストでは、基本ブロック内で起った複数の例外のうち、元のプログラム中の最初に存在する例外発生検出命令の例外ハンドラに分岐すればよい。

【００６７】

一方、プレディケート付き命令を実行できないアーキテクチャでは、条件分岐によって例外発生検出命令によるガードが実装される。そのため、例外発生を検出した場合の分岐先に、必要な副作用の実行と例外検出を行うための補償コードが必要となる（図３のステップ３０４及び図６参照）。補償コードは、コードスケジューリング最適化が終了して、演算間の順序が決定した後に求める。

【００６８】

図１３は、所定のコード（図１３（ａ））に対して、必要な補償コードを補った様子（図１３（ｂ））を説明する図である。また、図１４は、図１３に示すコードの順序制約を示す図である。

図１３において、下図でtestは例外発生検出命令、testの間のアルファベットは副作用のある演算を表すとする。副作用と例外発生検出命令の間には、上述した推移的な制御依存がある。すなわち、副作用AB_1はtest（Ａ）とtest（Ｂ）の前に実行することはできない。

【００６９】

図１３（ａ）に示す例外発生検出命令と副作用の順番で命令が実行されるので

あれば、補償コードは不要である。すなわち、例外発生検出命令から、ガードされている演算を全てスキップするように条件分岐を生成すればよい。

これに対し、コードスケジューリング最適化によって、図１３（ｂ）に示すように例外発生検出命令と副作用が配置された場合、例外発生検出命令及び副作用の関係は図１３（ａ）の場合のように単純ではない。すなわち、例外発生が検出されたパスで必要な補償コードは、図１４（ａ）に示す元のコードにおける例外及び副作用の順序制約を溯った場合には実行されていなければならない。しかし、かかる補償コードは、実際には例外が検出されるまで実行されておらず、例外発生検出命令の分岐から復帰した後も実行されない。

【００７０】

例えば、test（Ｂ）で例外が検出された場合、この時点ではtest（Ａ）、A_0が実行されていない。図１３（ａ）に示す元のコードではtest（Ａ）で例外が発生しなければ副作用A_0が実行されているので、図１３（ｂ）に示すように、test（Ａ）でA_0をガードしたコードを挿入する。同様に、test（Ｃ）で例外が検出された場合、この時点では副作用AB_1がまだ実行されていないので、図１３（ｂ）に示すように、AB_1を補償コードとして生成する。

【００７１】

もし、test（Ａ）とtest（Ｂ）における例外が例外ハンドラから見て区別できない場合は、補償コードをさらに少なくすることができる。例えば、図１３（ａ）に示す元のコードにおいて、A_0の位置に副作用がなく、test（Ａ）とtest（Ｂ）が同じ種類の例外であれば、順序制約は図１４（ｂ）のようになる。この場合、図１３（ｂ）に示した順序でコードが配置されても、副作用A_0が存在しないので、test（Ｂ）の補償コードによりtest（Ａ）における例外の発生を検出する必要はない。

【００７２】

次に、本実施の形態を用いた、数種類の対象アーキテクチャ（プログラムを実行する計算機）に対応したコード生成の例について説明する。

まず、ｘ８６アーキテクチャ（米インテル社製のプロセッサにおける８０８６、８０２８６、３８６アーキテクチャ）を対象としたコード生成例を説明する。

Ｘ８６アーキテクチャは、ユーザー定義のフラグレジスタ及びプレディケートを持たない。Ｘ８６アーキテクチャ上の主要なオペレーティングシステムであるＷｉｎ３２、ＯＳ／２、Ｌｉｎｕｘでは、仮想メモリ上の０番地から始まるページには読み出し及び書き込み保護が設定されているので、Ｊａｖａ言語のNullPointerExceptionの判定にはハードウェアによるアクセス違反例外の例外ハンドラで検出することが可能である。したがって、例外発生可能命令を投機実行しない場合は、NullPointerExceptionの検出命令に対しては実際にコード生成をしなくてよい。

【００７３】

しかし、ソフトウェアによる投機的実行を行う場合は、例外発生可能命令をガードする必要があるので、NullPointerExceptionの検出命令に対応する命令を明示的に生成しなければならない。Ｗｉｎ３２プラットフォームを対象とした場合には、Ｊａｖａバイトコードのiaload命令の投機的実行に必要な基本操作は、図１５に示すように実装される。

図１５において、arrhは配列オブジェクトのヘッダを格納するレジスタ、idxは配列アクセスの添字の値を格納するレジスタである。eHandlerは元々の例外ハンドラの先頭を示す。flHandlerは補償コードの実行及び代表フラグの設定を行うコードの先頭を示す。また、selHandlerは最初に発生した例外を調べて元のeHandlerに分岐するコードの先頭を示す。

【００７４】

iaload命令の投機的実行に必要な基本操作が図１５に示すように実装されることは、投機的に実行されるメモリアクセス命令がデータ依存及び副作用だけを考慮したＤＡＧのクリティカルパス上にあった場合に、投機的実行によってクリティカルパスの長さを伸ばしてしまう可能性があることを意味している。

【００７５】

この問題を回避するため、以下のような手法を取る。

すなわち、データ依存と副作用だけを考慮して得られるＤＡＧについて、各頂点の自由度（slackness）を計算する。自由度とは、当該頂点を含む経路に対して、全体のクリティカルパス長を伸ばすことなく、後何サイクルの演算を挿入す

ることができるかを表す数字である。投機的実行を行う例外発生可能命令を選択する場合に、自由度が条件分岐命令のコストよりも小さければ、その例外発生可能命令の投機的実行を行わない。

なお、各頂点の自由度は図１６に示すようにして計算される。

【００７６】

次に、ＰｏｗｅｒＰＣアーキテクチャ（米国ＩＢＭ社、米国アップルコンピュータ社、米国モトローラ社が開発したＣＰＵであるＰｏｗｅｒＰＣにおけるアーキテクチャ）を対象としたコード生成例を説明する。

ＰｏｗｅｒＰＣアーキテクチャは、分岐条件を記憶する専用のコンディションレジスタを持つ。上述したＸ８６アーキテクチャのコード生成例と同様に、本手法を適用することによって、コードスケジューラが例外発生検出命令の比較演算と通常の演算における計算との並列性を利用することが可能になる。ＰｏｗｅｒＰＣアーキテクチャでは、比較演算と分岐命令の間に３サイクル以上のレイテンシを置けば、分岐命令は命令フェッチユニットで処理される。このため見かけ上、分岐命令のオーバーヘッドが０になる（０サイクル分岐）。したがって、例外発生可能命令及び例外発生検出命令の並列性を抽出することで、クリティカルパス長を伸ばすことなく演算パイプラインの充填率を上げることが可能になる。ＡＩＸ（米国ＩＢＭ社のＵＮＩＸ）プラットフォームを対象とした場合、Ｊａｖａバイトコードのiaload命令の投機的実行に必要な基本操作は図１７に示すように実装される。

図１７において、arrh、flHandler、selHandler、idxの意味は上述したｘ８６アーキテクチャにおける図１５の場合と同様である。lenは、配列の長さを格納するレジスタである。

【００７７】

次に、ＩＡ－６４アーキテクチャを対象としたコード生成例を説明する。

ＩＡ－６４アーキテクチャは、プレディケート付き命令フォーマットを持つ。これにより、命令がプレディケートでガードされており、プレディケートの真偽値にしたがって命令が実行される。したがって、先に述べたようにガード付きで表現されたＤＡＧをプレディケートで実装することによって、例外が発生した場

合の補償コードを生成せずに済ませることができる。プレディケートで実装されたコードに対しては、コードスケジューリング及びコード生成を直接適用することができる。ＷＩＮ６４プラットフォームを対象とした場合、Ｊａｖａバイトコードのiaload命令の投機的実行に必要な基本操作は図１８に示すように実装される。

図１８において、arrh、idx、eHandler、selHandlerの意味は上述したｘ８６アーキテクチャにおける図１５の場合と、lenの意味は上述したＰｏｗｅｒＰＣアーキテクチャにおける図１７の場合と、それぞれ同様である。プレディケートの論理積は、並列比較を用いることによって、余分な命令オーバーヘッドを伴わずに実現できる。

【００７８】

以上のように、従来の技術ではハードウェアによるサポートが必要であった例外発生可能命令の投機的実行が、本実施の形態によってソフトウェアで実装可能となる。これにより、Ｊａｖａのような例外発生可能命令の順序制約が厳密である言語において、コードスケジューラによる命令レベルの並列性の抽出を妨げる順序制約を解消することが可能になった。

また上記のように、本実施の形態は、プレディケートを持たないプロセッサに実装した場合でも、例外ハンドラへの条件分岐命令を基本ブロックあたり２命令追加するだけで、例外発生検出命令の付加的なオーバーヘッド無しに例外発生可能命令の投機的実行を実現できる。

【００７９】

【発明の効果】

以上説明したように、本発明によれば、例外発生可能命令の他の命令への先行制約をソフトウェア的に緩和することによって、例外発生可能命令の投機的実行を実現でき、例外発生可能命令を含むプログラムの命令レベル並列性を効果的に得ることが可能となる。

【図面の簡単な説明】

【図１】　本発明の実施の形態におけるコンパイラの構成を説明する図である。

【図２】　本実施の形態における四つ組中間コード最適化部の構成を説明するブロック図である。

【図３】　本実施の形態におけるＤＡＧ編集部による編集処理の全体的な流れを説明するフローチャートである。

【図４】　図３のステップ３０１における例外による順序制約の緩和処理の詳細を説明するフローチャートである。

【図５】　図３のステップ３０２における例外ハンドラへの条件分岐のマージ処理の詳細を説明するフローチャートである。

【図６】　図３のステップ３０４における投機的に実行される例外に対する補償コードの生成処理の詳細を説明するフローチャートである。

【図７】　本実施の形態の処理対象であるプログラムの中間コード及びその四つ組中間コードの例を示す図である。

【図８】　図７のプログラムの持つ制約をＤＡＧで表現した図である。

【図９】　図８のＤＡＧに対して編集を行った状態を示す図である。

【図１０】　本実施の形態の処理対象であるプログラムの中間コード及びその四つ組中間コードの他の例を示す図である。

【図１１】　図１０のプログラムの持つ制約をＤＡＧで表現した図である。

【図１２】　図１１のＤＡＧに対して編集を行った状態を示す図である。

【図１３】　所定のコードに対して、必要な補償コードを補った様子を説明する図である。

【図１４】　図１３に示すコードの順序制約を示す図である。

【図１５】　ｘ８６アーキテクチャにおいて、Ｊａｖａバイトコードのiaload命令の投機的実行を行うために必要な基本操作図の実装例を示す図である。

【図１６】　データ依存と副作用だけを考慮して得られるＤＡＧについて、各頂点の自由度を計算する手法を説明する図である。

【図１７】　ＰｏｗｅｒＰＣアーキテクチャにおいて、Ｊａｖａバイトコードのiaload命令の投機的実行を行うために必要な基本操作図の実装例を示す図である。

【図１８】　ＩＡ－６４アーキテクチャにおいて、Ｊａｖａバイトコードの

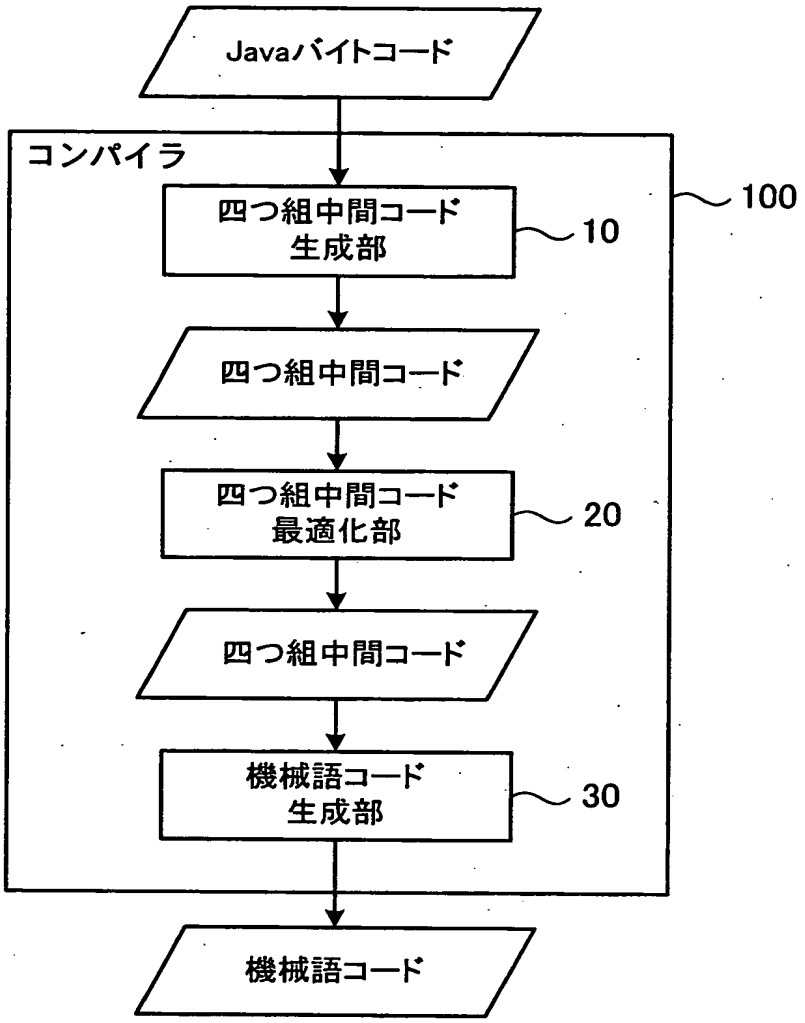
iaload命令の投機的実行を行うために必要な基本操作図の実装例を示す図である。

【符号の説明】

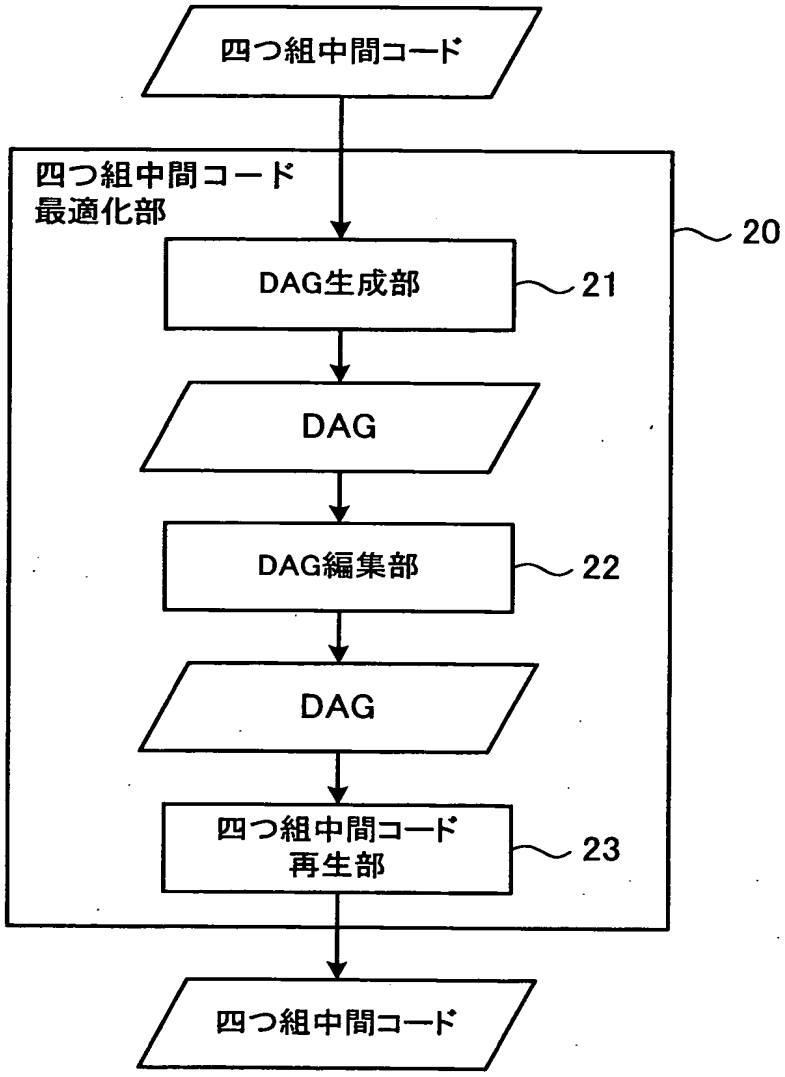
１０…四つ組中間コード生成部、２０…四つ組中間コード最適化部、２１…ＤＡＧ生成部、２２…ＤＡＧ編集部、２３…四つ組中間コード再生部、３０…機械語コード生成部

【書類名】　　図面

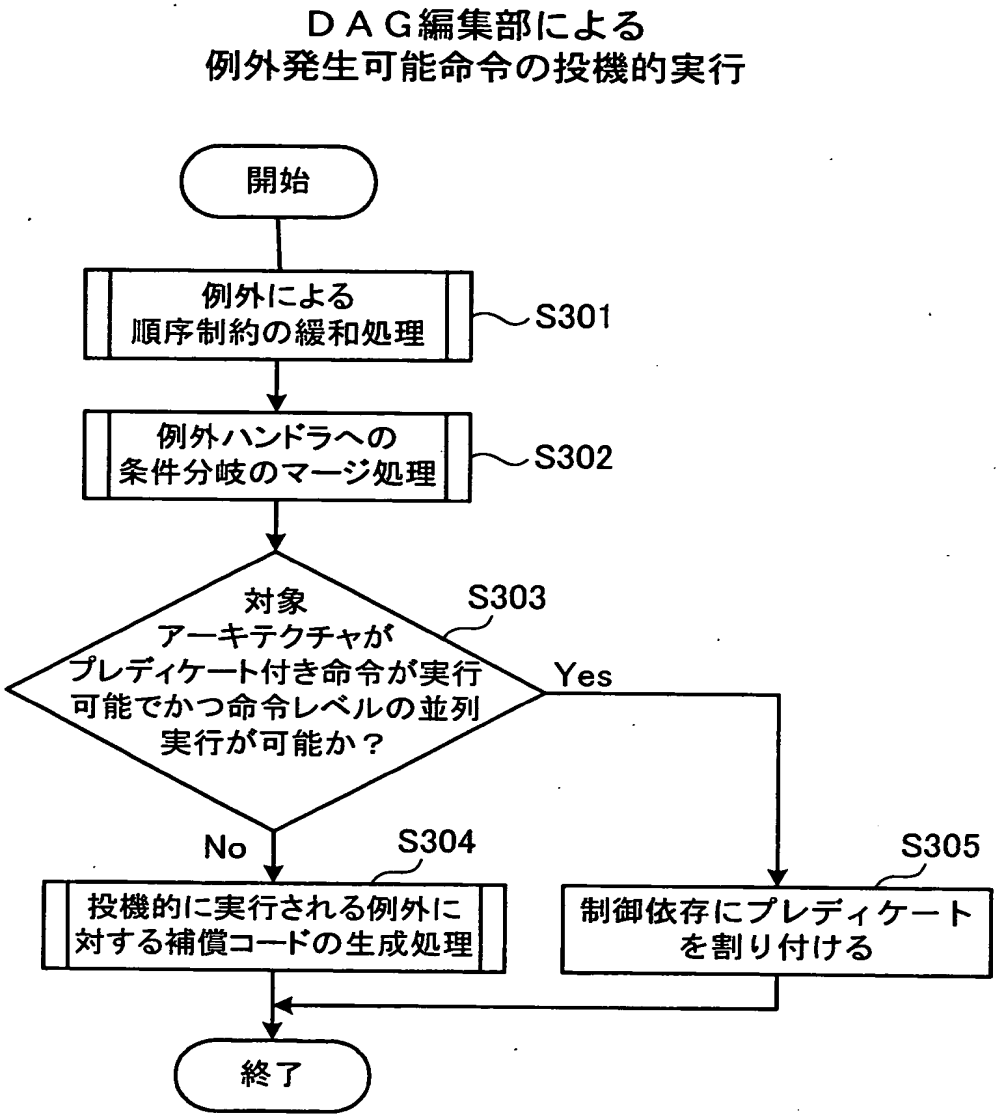
【図１】


Javaバイトコード
コンパイラ
100
四つ組中間コード
生成部
10
四つ組中間コード
四つ組中間コード
最適化部
20
四つ組中間コード
機械語コード
生成部
30
機械語コード

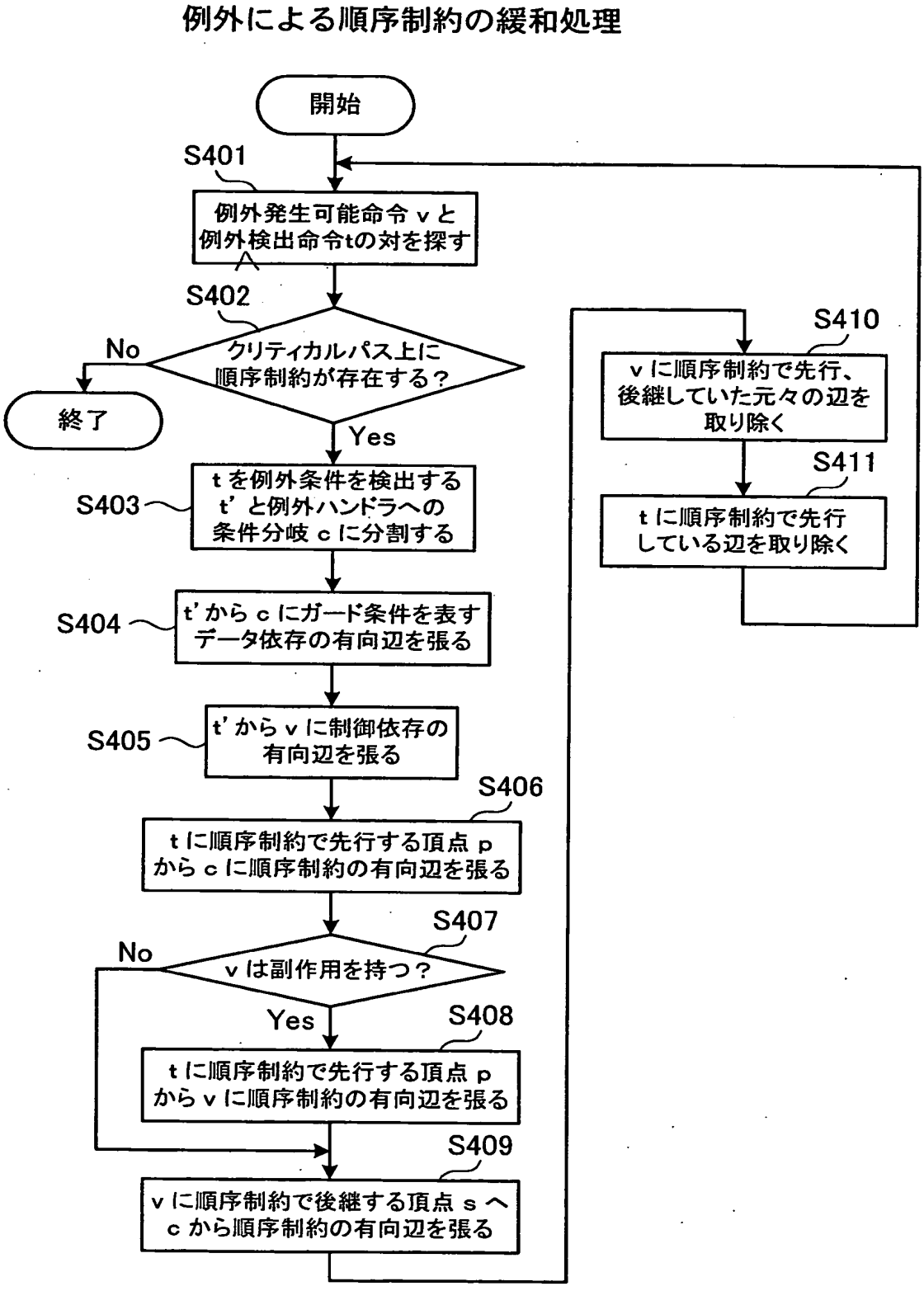
【図２】


四つ組中間コード
四つ組中間コード
最適化部
20
DAG生成部
21
DAG
DAG編集部
22
DAG
四つ組中間コード
再生部
23
四つ組中間コード

【図３】

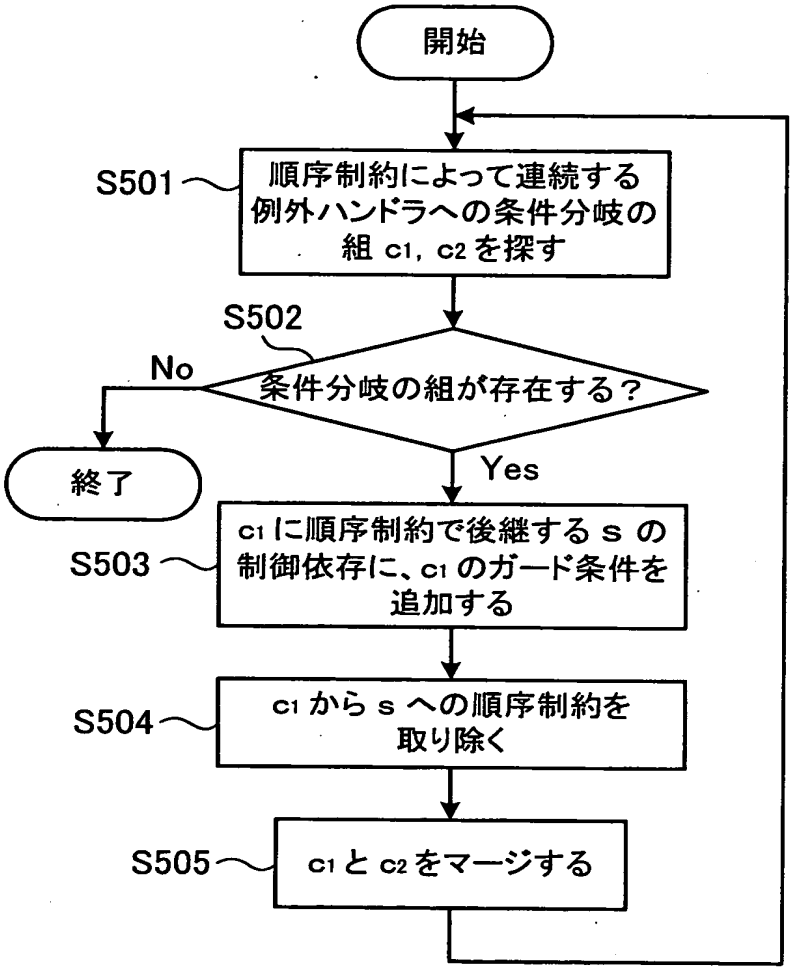

ＤＡＧ編集部による
例外発生可能命令の投機的実行
開始
例外による
順序制約の緩和処理
S301
例外ハンドラへの
条件分岐のマージ処理
S302
対象
アーキテクチャが
プレディケート付き命令が実行
可能でかつ命令レベルの並列
実行が可能か？
S303
Yes
No
S304
S305
投機的に実行される例外に
対する補償コードの生成処理
制御依存にプレディケート
を割り付ける
終了

【図４】

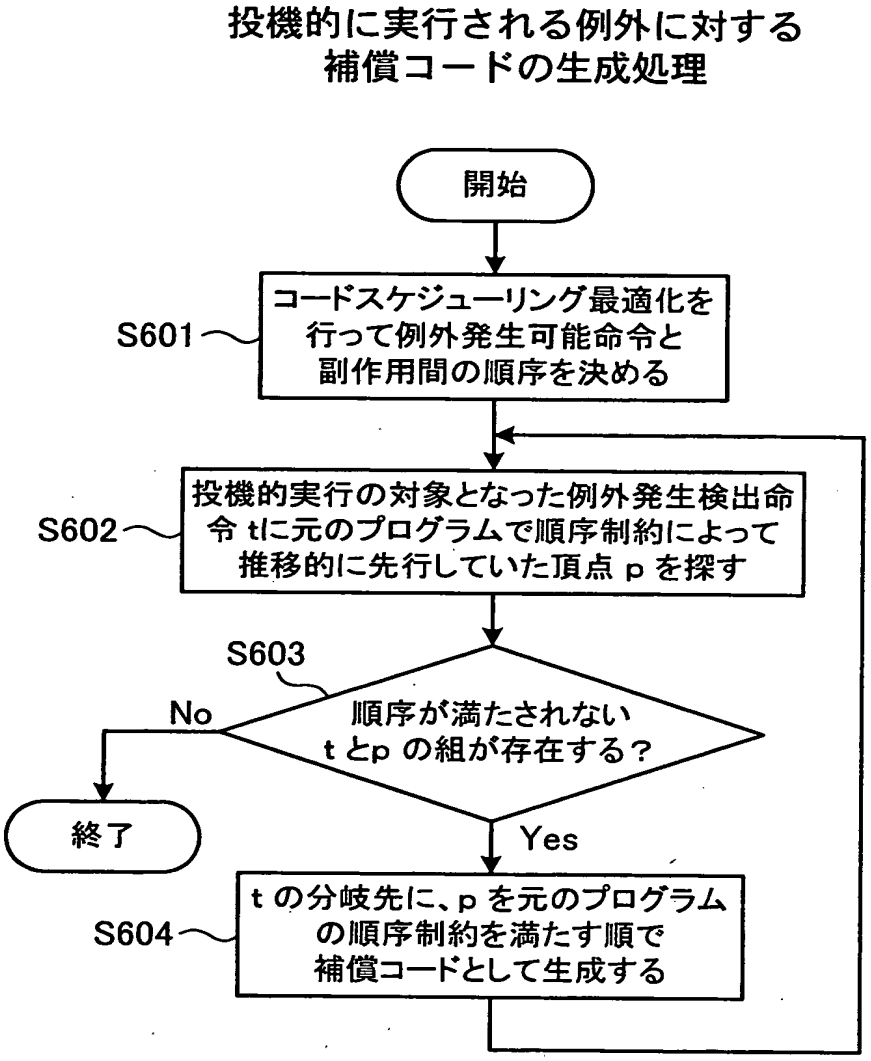

例外による順序制約の緩和処理
開始
S401
例外発生可能命令ｖと
例外検出命令tの対を探す
S402
クリティカルパス上に
順序制約が存在する？
No
終了
Yes
S403
ｔを例外条件を検出する
ｔ’と例外ハンドラへの
条件分岐ｃに分割する
S404
ｔ’からｃにガード条件を表す
データ依存の有向辺を張る
S405
ｔ’からｖに制御依存の
有向辺を張る
S406
ｔに順序制約で先行する頂点ｐ
からｃに順序制約の有向辺を張る
S407
No
ｖは副作用を持つ？
Yes
S408
ｔに順序制約で先行する頂点ｐ
からｖに順序制約の有向辺を張る
S409
ｖに順序制約で後継する頂点ｓへ
ｃから順序制約の有向辺を張る
S410
ｖに順序制約で先行、
後継していた元々の辺を
取り除く
S411
ｔに順序制約で先行
している辺を取り除く

【図５】

例外ハンドラへの条件分岐のマージ処理



【図６】


投機的に実行される例外に対する
補償コードの生成処理
開始
S601
コードスケジューリング最適化を
行って例外発生可能命令と
副作用間の順序を決める
S602
投機的実行の対象となった例外発生検出命
令 tlに元のプログラムで順序制約によって
推移的に先行していた頂点 p を探す
S603
No
順序が満たされない
t とp の組が存在する？
終了
Yes
S604
t の分岐先に、p を元のプログラム
の順序制約を満たす順で
補償コードとして生成する

【図７】

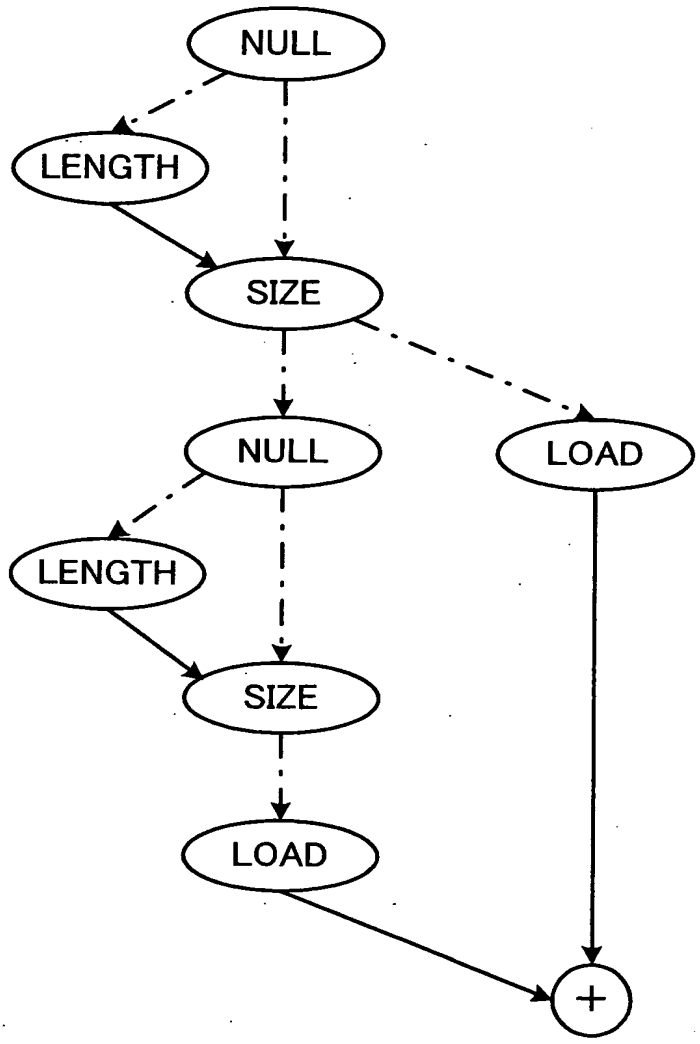
(A)

```
double    test    (double a[], double b[], int i, int j)   {
   return  a[i]    + b[j];
}
```

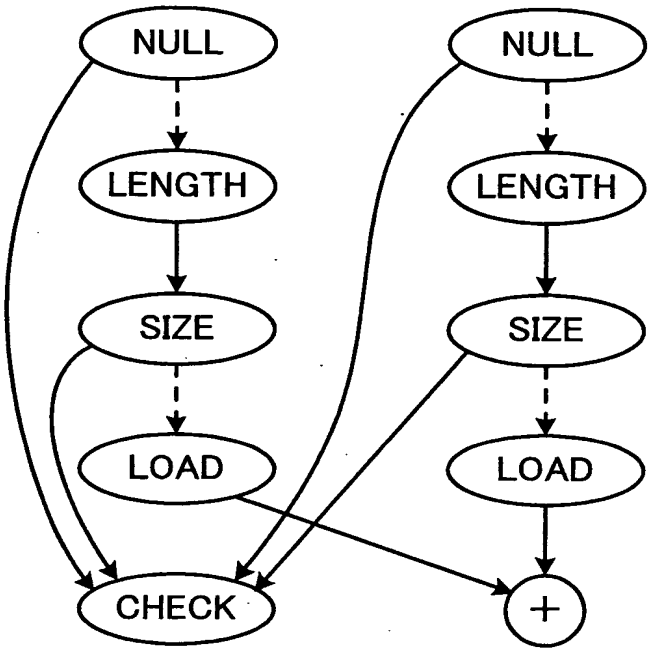
(B)

NULL		a
LENGTH	t=	a
SIZE		t, j
LOAD	x=	a, j
NULL		b
LENGTH	t=	b
SIZE		t, k
LOAD	y=	b, k
ADD	z=	x, y

【図８】



【図９】



【図１０】

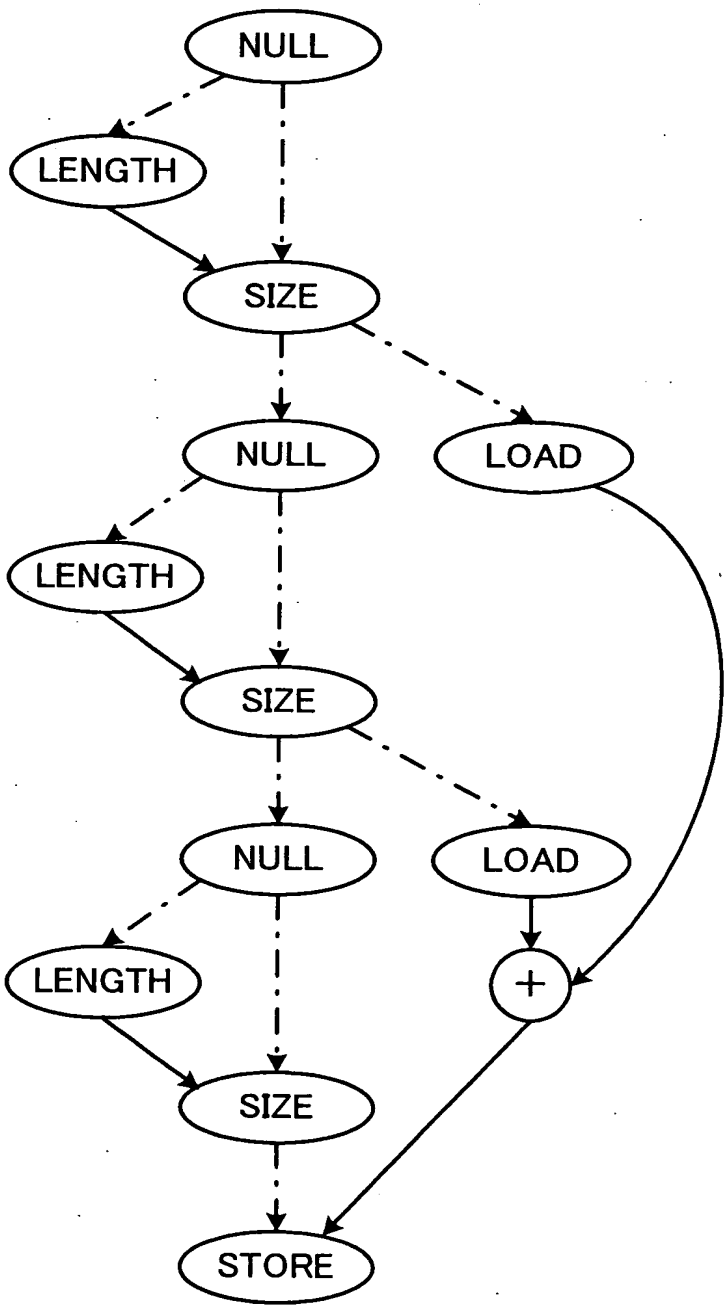
(A)

```
void     test    (double a[], double b[], double c[],
                  int i, int j, int k)  {
   a[i] = b[j]  +  c[k];
}
```

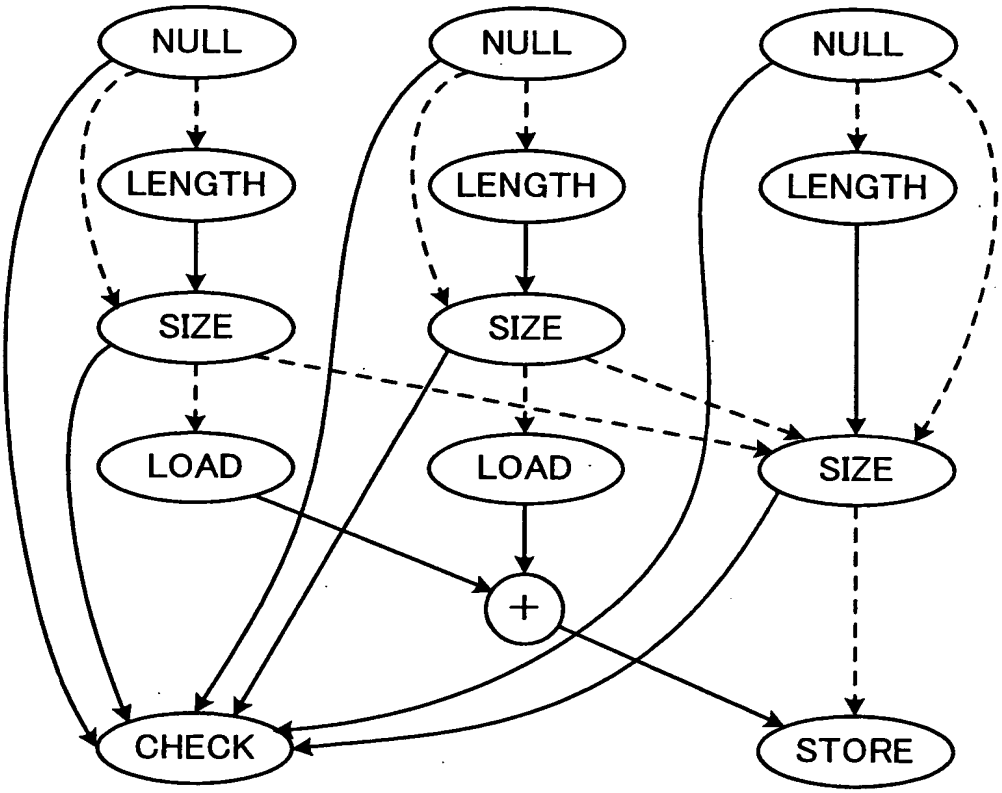
(B)

NULL		b
LENGTH	t=	b
SIZE		t, j
LOAD	x=	b, j
NULL		c
LENGTH	t=	c
SIZE		t, k
LOAD	y=	c, k
ADD	z=	x, y
NULL		a
LENGTH	t=	a
SIZE		t, i
STORE		a, i, z

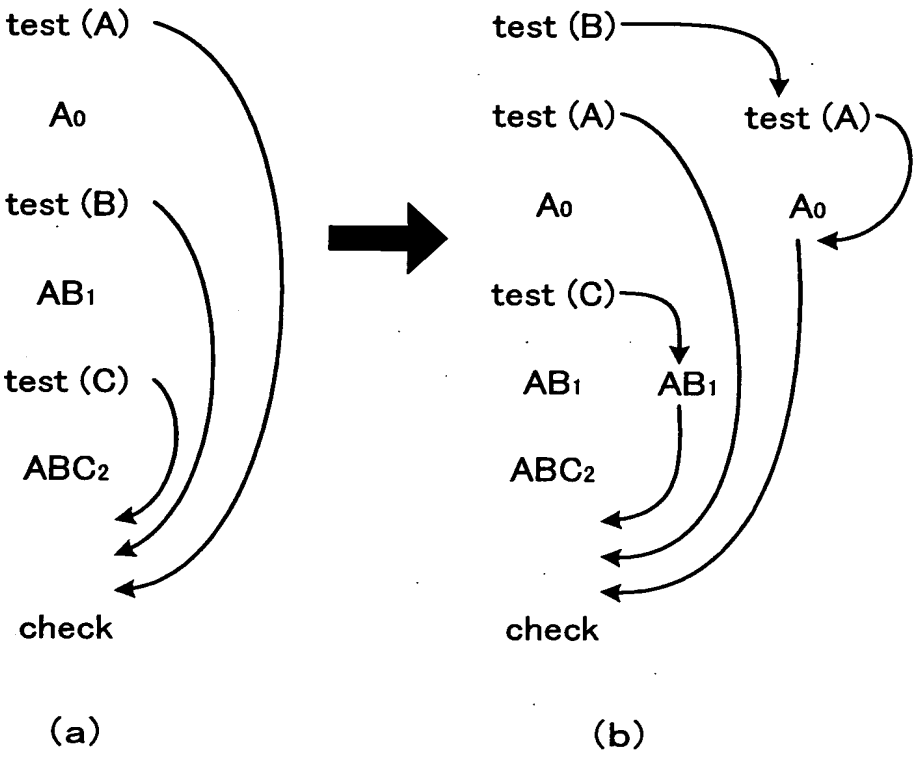
【図１１】


NULL
LENGTH
SIZE
NULL
LOAD
LENGTH
SIZE
NULL
LOAD
LENGTH
+
SIZE
STORE

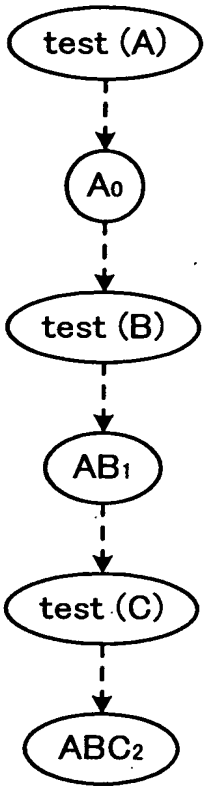
【図１２】


NULL
NULL
NULL
LENGTH
LENGTH
LENGTH
SIZE
SIZE
LOAD
LOAD
SIZE
+
CHECK
STORE

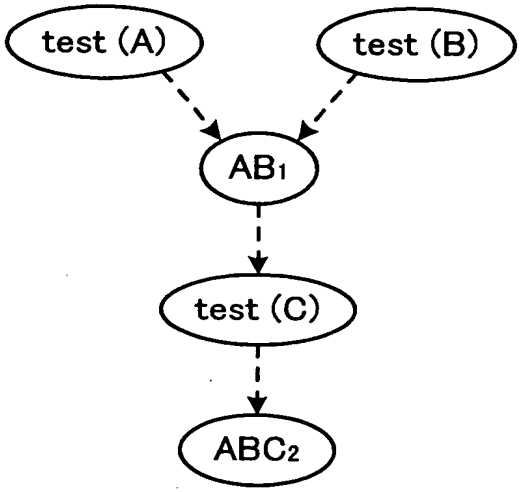
【図１３】


test (A)
A0
test (B)
AB1
test (C)
ABC2
check
(a)
test (B)
test (A)
test (A)
A0
A0
test (C)
AB1
AB1
ABC2
check
(b)

【図１４】



(a)



(b)

【図１５】

	NullPointerException	ArrayIndexOutOf BoundsException	代表フラグのチェック
通常実行	なし	cmp idx, [arrh] jae eHandler	なし
投機的実行	cmp arrh, 00h jeq flHandler	cmp idx, [arrh] jae flHandler	cmp flag, 00h jne selHandler

【図１６】

入力

V: 四つ組の演算子に対応する頂点の集合

E: 演算子間の依存関係に対応する辺の集合

$G(V, E)$: プログラムを表すDAG

$\tau \in V$: 先行する演算子を持たない全ての演算子に先行する仮想的な頂点（トップ）

$\perp \in V$: 後継する演算子を持たない全ての演算子に後継する仮想的な頂点（ボトム）

$cy(v)$: 頂点vの演算を実行するために必要なマシンサイクル数。

$cp(G)$: 例外による順序制約を無視したときのプログラムを表すDAGのcritical path長

手続き（頂点の自由度の計算）

$lb(v)$: 例外による順序制約を無視したときの頂点vのボトムからのレベル

1. 後継する頂点がない場合は$lb(v)=0$.
2. 後継する頂点がある場合は、vに例外による順序制約以外の辺で後継する頂点の集合を$s \in succ(v)$と表すと、

$$lb(v) = \max(lb(s)) + cy(v).$$

$lt(v)$: 例外による順序制約を無視したときの頂点vのトップからのレベル

1. 先行する頂点がない場合は$lt(v)=0$.
2. 先行する頂点がある場合は、vに例外による順序制約以外の辺で先行する頂点の集合を$p \in pred(v)$と表すと、

$$lt(v) = \max(lt(p) + cy(v)).$$

$sl(v)$: 例外による順序制約を無視したときの頂点vの自由度

$$sl(v) = cp(G) - lt(v) - lb(v).$$

【図１７】

	NullPointerException	ArrayIndexOutOf BoundsException	代表フラグのチェック
通常実行	tlei arrh, 00h	lw len, [arrh] tlle idx, len	なし

投機的実行	cmp arrh, 00h
	beq flHandler
		lw len, [arrh]
		cmpl idx, len
		bgt flHandler
			bne cr2, selHandler

【図１８】

	NullPointerException	ArrayIndexOutOf BoundsException	代表フラグのチェック
通常実行	dmp.ne p1, p2 = 00h, arrh (p1) br eHandler	ld8 len = [arrh] cmp.ltu p1, p2 = idx, len (p1) br eHandler	なし
投機的実行	cmp.ne.and p1,p2 = 00h, arrh	ld8 len = [arrh] cmp.ltu.and p1, p2 = idx, len	(p1) br selHandler

【書類名】　　要約書

【要約】

【課題】　例外発生可能命令の他の命令への先行制約をソフトウェア的に緩和し、例外発生可能命令を含むプログラムの命令レベル並列性を効果的に得ることを可能とする。

【解決手段】　コンパイラにおいて、処理対象のプログラムの四つ組中間コードを解析し、ＤＡＧを生成するＤＡＧ生成部２１と、生成されたＤＡＧを編集し、例外による演算子の順序制約を緩和するＤＡＧ編集部２２と、編集後のＤＡＧの構造を反映させた四つ組中間コードを生成する四つ組中間コード再生部２３とを備える。そして、ＤＡＧ中から、例外発生可能命令と例外発生検出命令とを検出し、検出された例外発生検出命令を、例外の発生条件を検出する第１の命令と、例外処理へ処理を条件分岐させる第２の命令とに分割し、第１の命令に対して、例外の発生条件を検出した場合に第２の命令に移行し、例外発生条件を検出しなかった場合に例外発生可能命令に移行するように依存関係を設定する。

【選択図】　　図２

認定・付加情報

特許出願の番号	特願２０００－３３１４２７
受付番号	５００００１４０４７５８
書類名	特許願
担当官	塩崎　博子　　１６０６
作成日	平成１２年１２月１３日

＜認定情報・付加情報＞

【特許出願人】
- 【識別番号】 390009531
- 【住所又は居所】 アメリカ合衆国１０５０４、ニューヨーク州　アーモンク　（番地なし）
- 【氏名又は名称】 インターナショナル・ビジネス・マシーンズ・コーポレーション

【代理人】
- 【識別番号】 100086243
- 【住所又は居所】 神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　大和事業所内
- 【氏名又は名称】 坂口　博

【代理人】
- 【識別番号】 100091568
- 【住所又は居所】 神奈川県大和市下鶴間１６２３番地１４　日本アイ・ビー・エム株式会社　大和事業所内
- 【氏名又は名称】 市位　嘉宏

【代理人】
- 【識別番号】 100106699
- 【住所又は居所】 神奈川県大和市下鶴間１６２３番１４　日本アイ・ビー・エム株式会社大和事業所内
- 【氏名又は名称】 渡部　弘道

【復代理人】 申請人
- 【識別番号】 100104880
- 【住所又は居所】 東京都港区赤坂５－４－１１　山口建設第２ビル６Ｆ　セリオ国際特許事務所
- 【氏名又は名称】 古部　次郎

【選任した復代理人】
- 【識別番号】 100100077

次頁有

認定・付加情報（続き）

【住所又は居所】　東京都港区赤坂５－４－１１　山口建設第２ビル
６Ｆ　セリオ国際特許事務所
【氏名又は名称】　大場　充

次頁無

出　願　人　履　歴　情　報

識別番号　　〔３９０００９５３１〕

１．変更年月日　　２０００年　５月１６日
　［変更理由］　名称変更
　　住　所　　アメリカ合衆国１０５０４、ニューヨーク州　アーモンク　（番地なし）
　　氏　名　　インターナショナル・ビジネス・マシーンズ・コーポレーション